UPC Performance Evaluation on a Multicore System

Damián A. Mallón, J. Carlos Mouriño, Andrés Gómez

Galicia Supercomputing Center
Santiago de Compostela, Spain
{dalvarez,jmourino,agomez}@cesga.es

Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Ramón Doallo
Computer Architecture Group
University of A Coruña, Spain
{taboada,cteijeiro,juan,basilio,doallo}@udc.es

Brian Wibecan

Hewlett-Packard
Nashua (NH), USA
brian.wibecan@hp.com

***Abstract*—As size and architectural complexity of High Performance Computing systems increases, the need for productive programming tools and languages becomes more important. The UPC language aims to be a good choice for a productive parallel programming. However, productivity is influenced not only by expressiveness of the language, but also by its performance. To assess the current UPC performance in high performance multicore systems, and therefore to help improve UPC developers future productivity, this paper provides an up-to-date UPC performance evaluation at various levels, evaluating two collective implementations, comparing their results with their MPI counterparts, and finally evaluating UPC and MPI performance in computational kernels. This analysis shows a path to optimize UPC collectives performance. This work also provides a performance snapshot of UPC vs the currently most popular choice for parallel programming, MPI. This snapshot, altogether with the UPC collectives analysis, shows that there is room for improvement and, besides its worse performance, UPC is suitable for a productive development of most HPC applications.**

I. Introduction

As multicore systems are increasing their popularity, UPC [1] has shown to be a good alternative to more traditional parallel programming models (e.g., message-passing, data parallel and shared-memory models), due to its Partitioned Global Address Space (PGAS) memory model. The PGAS memory model makes UPC especially suitable for hybrid shared/distributed memory architectures. Nevertheless, the transparent access to data affine to a remote thread usually introduces a significant overhead.

In order to reduce the number of these inefficient operations, the UPC collectives specification [2], which is part of the standard UPC specification [3], defines a set of data movements and computational operations commonly used in parallel applications. The implementation of these primitives, in a UPC collectives library, provides improvements in the programmability, as well as the locality (and hence the performance) in the data access.

Regarding UPC compilers, the most relevant open-source implementation is Berkeley UPC [4] (BUPC). BUPC provides its own collectives library (from now on "BCOL", Berkeley COllectives Library). However, it can use other collectives libraries, such as the UPC reference implementation (from now on "REF") [5] from Michigan Tech. University. Besides BUPC, there are also other commercial and open source UPC compilers, mainly HP UPC [6] and IBM UPC [7].

As the performance of the collectives operations is critical for overall performance, this paper presents an up-to-date performance evaluation of the two collective libraries aforementioned, comparing their results with the equivalent MPI operations performance. Due to the lack of suitable benchmarking tools for this task, a microbenchmarking suite has been developed as part of this work. Also, in order to provide a snapshot of UPC current performance in computational kernels, a second analysis has been done, using the NAS Parallel Benchmarks (NPB) [8] to compare UPC and MPI.

The rest of this paper is organized as follows. Section 2 discusses the related work on UPC performance evaluations. Section 3 presents the current implementations of UPC primitives and the different extensions and optimizations that have been proposed, as well as the microbenchmarking suite developed as part of this work. It also presents the NPB and the particular characteristics of their UPC implementation. Section 4 contains the results of our evaluation conducted on a multicore InfiniBand supercomputer. Finally, Section 5 concludes the paper.

II. Related Work

Previous UPC performance evaluations [9], [10], [11] have focused either on basic data movement primitives or whole applications. Data movement primitives include `upc_memget`, `upc_memput` and `upc_memcpy`, which process all data types as byte arrays (raw data). Regarding the evaluated applications, the most relevant ones are general problem-solving algorithms, such as matrix multiplication, Sobel edge detection, N-Queens, and the UPC version of the NPB. The application benchmarks allowed to measure overall results of UPC performance, whereas the microbenchmarking of memory primitives showed communication latencies and bandwidths.

A shared outcome from these studies is that UPC shows better performance when data locality and the use of private memory are maximized [12]. Collective primitives have not played an important role in these studies: research on this topic has been mostly restricted to different proposals for optimizing

or extending the standard collectives specification. Recently, the introduction of MPI-like optimizations on UPC collectives has been proposed and experimentally evaluated in [13]. Nevertheless, these works only present relative comparisons between UPC collective libraries, measuring percentages of improvement, without characterizing their real performance, so the comparison with other implementations is not possible. Moreover, these three studies are restricted to the use of a small number of threads (up to 16) on a cluster of dual processor nodes.

The analysis of the throughput of collectives primitives is useful for selecting the implementation that obtains the highest performance in a given scenario, to detect inefficient implementations, and eventually to propose new algorithms in order to increase their performance. The lack of this analysis is the main motivation of this work. Also, this paper shows representative performance results of today's application setups, using a larger workload (NPB Class C) and a higher number of cores (128) than previous works.

III. Benchmarking Aspects

A. Implementations of UPC Collectives

The UPC collectives specification includes several primitives that implement common communication patterns, such as broadcast, scatter, gather, exchange or reduce. These primitives operate in the shared-memory space, which implies that the source and destination arguments of these primitives are pointers to shared-memory locations. In order to improve the functionality of the standard collective specification, different extensions and optimizations have been proposed. The most relevant ones are Value-based Collectives [14], One-sided Collectives [15] and Variable-sized Data Blocks Collectives [16]. Value-based Collectives optimize communications of single-valued variables, which can be either on private or shared memory. The One-sided approach defines communications in a single direction, with an active and a passive peer for each communication, thus simplifying synchronizations in data transfers. Variable-sized Data Blocks Collectives provide a more flexible set of collectives that define custom message sizes and source/destination pointers for each communication peer, allowing non-contiguous data movements and the transmission of data from private memory.

An additional project on UPC collectives is the definition of sets of threads called "teams", which allow a collective primitive to be called by a subset of all available threads [17], similarly to MPI communicators. Another active line is the research on extensions of the UPC memory copy library [18], that aims to suppport the efficient implementation of asynchronous communications and non-contiguous data transfers.

The basis of UPC collectives primitives are data transfers between threads, which can be implemented either with bulk data transfers, using UPC functions such as `upc_memcpy`, or relying on the collective implementation provided by an underlying communication library. Regarding the two UPC collective libraries evaluated in this study, the BUPC collectives library (BCOL) [4] relies on a low-level communication library (GASNet), whereas the UPC reference implementation (REF) is implemented with `upc_memcpy` operations.

From version 2.6.0 of the BUPC compiler, the former linear flat-tree implementation of collectives has been replaced by a binomial tree communication pattern, which organizes data transfers in a logarithmic number of steps, thus reducing memory and network contention.

Regarding REF, the implementation of its collectives primitives is based on a fully parallel flat-tree algorithm, so that they are all performed in only one step. Two different approaches can be used in REF collectives primitives: pull and push. Both techniques are based on `upc_memcpy`, and their distinguishing feature is the active side in the communications. In the pull approach, each destination thread copies its corresponding data from the source thread in parallel, while in the push approach each source thread copies its data to all the destination threads. The selection of a pull or a push approach has to provide a fair distribution of the communication overhead for each collective primitive. Thus, in broadcast and scatter a pull implementation is better because it makes the destination threads copy the data in parallel from the source thread, whereas the push approach maximizes parallelism in the gather collective. In this study, the most efficient approach has been selected for each REF collective primitive evaluated.

B. UPC Microbenchmark Suite

As there is a lack of suitable benchmarks for our purposes, we have implemented our own UPC collectives suite, which is similar to the Intel MPI collectives benchmarks (previously known as Pallas MPI) [19]. This suite has been designed to measure the performance of every collective primitive through a single call to a generic benchmarking function, which tests the performance of the primitive in a range of message sizes. To do this, each collective function is identified by a predefined integer, and this identifier with the minimum and maximum data sizes are passed as arguments to the generic collective benchmark call, that returns the performance results for the given number of threads in each execution. By using this suite, it is possible to characterize the performance of all collectives primitives present in the UPC specification, and also for raw memory copies. However, only five collectives primitives have been selected for our evaluation: broadcast, scatter, gather, exchange and reduce.

In order to avoid the issues that might arise when microbenchmarking communications performance, the benchmarking suite has been designed following most of the guidelines presented in [20]. For example, the UPC_{IN,OUT}_ALLSYNC (strict) synchronization mode has been used in all collective calls (in UPC the synchronization mode is passed to each collective primitive call as a function parameter). The main goal of this approach is to characterize the maximum overhead that the synchronization can impose in a collective primitive operation. Furthermore, the results have been obtained using cache invalidation in order to avoid the influence of cache reuse. This technique has been implemented using new

dynamically allocated buffers for each primitive call, without reuse. The design of the tests ensures the correctness of the results. However, simple sanity checks are also performed here. The performance results of UPC collectives obtained with our microbenchmark suite are discussed in the next section.

C. NAS Parallel Benchmarks

The NPB comprises of a set of kernels and pseudo-applications that reflect different kinds of relevant computation and communication patterns used by a wide range of applications, which makes them the de facto standard in parallel performance benchmarking. The NPB evaluated are: CG (an iterative equation solver), EP (an embarrassingly parallel code that assesses the floating point performance), FT (a series of 1-D FFTs on a 3-D mesh), IS (a large integer sort) and MG (a simplified multigrid kernel that performs both short and long distance communications). Each kernel has several workloads to scale from small systems to supercomputers.

NPB-MPI are implemented using Fortran, except for IS which is programmed in C. The fact that the NPB are programmed in Fortran has been considered as cause of a poorer performance of NPB-UPC [21], due to better backend compiler optimizations for Fortran than for C.

Most of the NPB-UPC kernels, developed at the George Washington University, have been manually optimized through techniques that mature UPC compilers should handle in the future: privatization, which casts local shared accesses to private memory accesses, avoiding the translation from global shared address to actual address in local memory, and prefetching, which copies non-local shared memory blocks into private memory.

IV. UPC Performance Results

The testbed used in this work is the Finis Terrae supercomputer [22], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory. The InfiniBand HCA is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). For the evaluation 8 nodes have been used (up to 128 cores). The number of cores used per node in the NPB performance evaluation is $\lceil n/8 \rceil$, n being the total number of cores used in the execution. $\lceil n/8 \rceil$ consecutive threads run in the same node.

The OS is SUSE Linux Enterprise Server 10 with kernel version 2.6.16 and OFED 1.3.1. The MPI implementation is HP MPI 2.2.5.1 with InfiniBand Verbs (IBV) for internode communication, and the shared memory transfers for intranode communication. The UPC compiler is Berkeley UPC 2.6 for the collective performance analysis, and Berkeley UPC 2.8 for the NPB performance results. Both versions use the IBV driver for distributed memory communication, and pthreads within a node for shared memory transfers. The MPI and UPC backend compiler is the Intel 10.1.012 for the collective tests, and Intel 11.0.069 for the NPB tests, both versions with -O3 flag.

A. UPC Collective Primitives Performance

Figure 1 shows latency (total communication overhead) for small messages (where it is more important than bandwidth) and, for long messages, aggregated bandwidths of UPC collectives for BUPC using two UPC collective implementations, BCOL and REF. The main difference between the collective implementations is that REF uses flat-tree communication algorithms, whereas BCOL resorts to binomial trees. The data size shown in the figures is the size of the data used in the collective operation at the root thread, or in any thread in a non-rooted operation. Thus, in a 1 MB scatter primitive (rooted operation) (1 MB)/*THREADS* of data is sent to each thread (*THREADS* is the number of UPC threads involved in the collective operation).

The aggregated bandwidth metric includes the minimum number of bytes that need to be transferred in the collective primitive operation, thus allowing us to compare the performance obtained using different numbers of threads (performance scalability). It has been calculated as *data_size / latency* for the scatter and gather measures, and as *THREADS * data_size / latency* for the broadcast and exchange results. The measured latency includes the synchronization barriers.

Each plot shows the performance of the two collective implementations (BCOL and REF) using 32 threads. The results have been obtained using 2, 4 and 8 nodes, which means using 16, 8 or 4 UPC threads per node, respectively. Table I presents a summary of the results of the reduce collective, showing latencies.

The first plot shows the performance of the broadcast collective. REF obtains the best latency. Binomial trees have a $O(log_2 N)$, whereas flat trees have a $O(N)$. However, in binomial trees almost half of the nodes are a source for two messages, whereas in flat trees there is only one source for all messages. Due to that, for small messages it does not compensate the use of binomial trees, because despite its lower complexity they have a bigger overhead. For long messages BCOL shows the best performance. The REF results are poor, as they involve several internode transfers, which are an important performance bottleneck for a flat-tree algorithm. BCOL outperforms their REF counterpart, emphasizing the fact that the minimization of internode transfers improves the performance of the collectives. From the analysis of the results it can be derived that there is not much difference between using 8 nodes with 4 threads per node, and using 4 nodes with 8 threads per node. The use of 2 nodes, and hence 16 threads per node, maximizes the number of intranode transfers, which benefits from the flat-tree algorithm of REF, whereas it harms BCOL performance.

The second graph presents the results of UPC scatter. It shows that the best performance has been obtained by REF thanks to its parallel access to the source thread, which avoids synchronization steps and data buffering in intermediate threads. This is the opposite behavior to the broadcast, where BCOL obtains better results than REF. In this case the BCOL scatter (binomial tree) has to transfer additional data for all

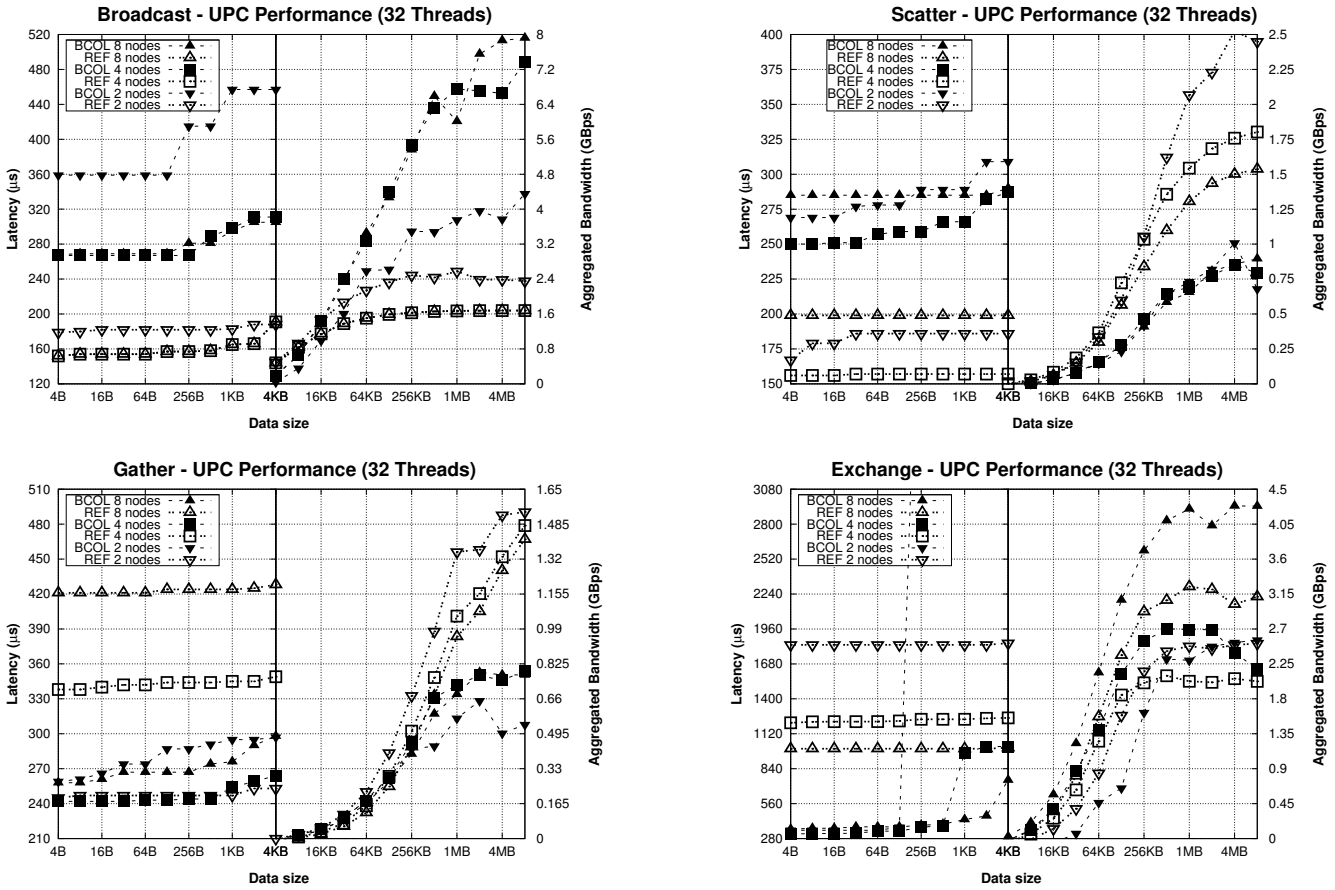

Figure 1. UPC operations performance

the leaves of a node (intermediate buffering). For example, in a 1 MB scatter to 8 threads using a binomial tree it is required that the source thread transfers 512 KB in the first step, the two threads with data (the source and the one that got the data in the first step) will transfer 256 KB in the second step, and finally, four threads will copy 128 KB to the leaves of the binomial tree. Thus, BCOL requires the movement of 1536 KB whereas REF only 1024 KB, which means an overhead in terms of extra data transferred of 50% of the data size considered in the primitive. Regarding a 32-thread operation, the additional data overhead is 153% of the data size considered in the primitive. Thus, for a 1 MB scatter, 1.53 MB of additional data are transferred. Therefore, REF scatter obtains higher throughput as it transmits the minimum amount of data without synchronization overheads. This efficient implementation allows REF to outperform BCOL. In this case the best results have been obtained using 2 nodes, and hence 16 threads per node, where the number of shared-memory transfers is maximized.

The third plot depicts the results of UPC gather. Similarly to scatter, REF usually outperforms BCOL for large messages. In fact, the evaluation of all the graphs of gather shows analogous conclusions to the previous primitive. However, the latency of REF gather increases with the number of nodes, showing that the management of several small internode messages introduces a significant overhead. Thus, for the smaller transfers it is better to use a binomial tree algorithm that maximizes the number of shared-memory transfers.

Table I
UPC REDUCE LATENCIES (32 THREADS)

	Data size					
	1KB			1MB		
Nodes	2	4	8	2	4	8
BCOL	242 μs	186 μs	213 μs	227 μs	305 μs	354 μs
REF	297 μs	249 μs	287 μs	3860 μs	3966 μs	3599 μs

The last plot in Figure 1 shows the exchange results. This primitive involves a more complex communication pattern than the preceding ones. The REF latency suffers from the use of few nodes, and increases significantly when 2 nodes are in use. However, as the message size increases, the performance gap narrows. Consequently, BCOL outperforms REF on 8 and 4 nodes, whereas REF outperforms BCOL on 2 nodes. Both BCOL and REF implementations benefit from the use of 8 nodes, compared to the use of 4 and 2, as for this non-rooted operation all threads are actively communicating, thus taking advantage of the highest number of nodes.

These results have been obtained with the UPC_{IN,OUT}_ALLSYNC synchronization modes. Preliminary results have shown that UPC collectives

Table II
UPC VS. MPI COLLECTIVES PERFORMANCE (32 THREADS/PROCESSES, 4 NODES, MPI = 100%)

	Broadcast		Scatter		Gather		Exchange/Alltoall	
Library \ Message size	1 KB	1 MB	1 KB	1 MB	1 KB	1 MB	1 KB	1 MB
MPI (GB/s)	0.0992	4.4013	0.0088	1.5360	0.0183	1.5627	0.0066	0.0971
BCOL	5%	86%	43%	44%	45%	45%	16%	89%
REF	9%	24%	72%	97%	3%	64%	13%	68%

can greatly reduce their communication overhead with UPC_{IN,OUT}_MYSYNC and UPC_{IN,OUT}_NOSYNC synchronization, especially for BCOL implementation and for small messages, thanks to the use of lighter synchronization modes (e.g., for a 32 byte BCOL broadcast with 32 threads and 2 nodes, MYSYNC and NOSYNC synchronization modes reduce the communication overhead, compared to ALLSYNC, in 47% and 77%, respectively).

Table I shows the latencies (in microseconds) of UPC reduce. The latency has been selected as performance measure instead of the bandwidth as the UPC reduce only involves the transfer of a primitive data type value per thread, independently of the number of elements being processed. The operation of the UPC reduce differs from the MPI reduction, which communicates all the local data. Thus, the reduction of an array of 10 elements per thread/process returns a scalar result in UPC and a 10 element result array for MPI. The operation used in the microbenchmarking is the floating point addition of double precision data (doubles). Unlike the previous data movement collectives, reduce is a computational one, and therefore its UPC implementation is more intensive in computations than in communications. Thus, in this scenario, it can be concluded that the computation associated to a reduce call happens to be implemented more efficiently in BCOL reduce than in REF, because BCOL clearly outperforms REF especially for long messages.

From the analysis of the performance results presented in this section it can be concluded that: (1) there are significant performance differences between BCOL and REF; therefore, it is possible to increase UPC throughput by selecting the best collective library at runtime for each configuration and message size; (2) and it is possible to optimize collective operations minimizing the number of internode communications and using a flat-tree algorithm for shared-memory transfers on intranode communication.

B. UPC vs. MPI Collective Performance Analysis

This subsection presents a comparative analysis of the performance of the UPC collectives and their MPI counterparts. The benchmarking software used is the UPC microbenchmarking suite presented in previous sections and the Intel MPI Benchmarks. As MPI collectives have been thoroughly optimized for years, the gap between MPI and UPC collectives performance can be considered a good estimate of the quality of a UPC implementation. However, UPC will not always lag behind MPI, as it is expected that UPC collectives outperform MPI when shared-memory transfers are involved. Table II shows the relative performance of UPC compared to MPI (HP-MPI v2.2.5.1), where UPC collectives throughput is shown as a percentage of the MPI performance. The reduce comparison is not shown, as the UPC reduce primitive has no equivalent operation in MPI (MPI reduce transfers an array instead of a single variable). These results have been obtained for two representative message sizes, 1 KB and 1 MB. An analysis of the results shows that the UPC performance is lower than MPI results. Furthermore, UPC suffers from higher start-up latencies than MPI, which means poor performance for 1 KB messages, especially for the broadcast. This comparative analysis of MPI and UPC collectives performance serves to assess that there is room for improvement in the implementation of the UPC collectives.

C. Performance Evaluation of UPC Kernels

The figure 2 shows NPB-MPI and NPB-UPC performance using both InfiniBand and shared memory communication. The left graphs show the kernel's performance in MOPS (Million Operations Per Second), whereas the right graphs present their associated speedups.

Regarding the CG kernel, MPI performs slightly worse than UPC using up to 32 cores due to the kernel implementation, whereas on 64, and especially on 128 cores, MPI outperforms UPC. Although UPC uses pthreads within a node, its communication operations, most of them point-to-point transfers with a regular communication pattern, are less scalable than MPI primitives.

EP is an embarrassingly parallel kernel, and therefore shows almost linear scalability for both MPI and UPC. The results in MOPS are approximately 6 times lower for UPC than for MPI due to the poorer UPC compiler optimizations. EP is the only NPB-UPC kernel that has not been optimized through prefetching and/or privatization, and the workload distribution is done through a `upc_forall` function, preventing more aggressive optimizations.

The performance of FT depends on the efficiency of the exchange collective operations. Although the UPC implementation is optimized through privatization, it presents significantly lower performance than MPI. In this kernel UPC is limited by its single thread performance, due to the lack of aggressive optimizations, and not by its communications. This why the UPC results, although significantly lower than MPI in terms of MOPS, show higher speedup than MPI. This is a communication-intensive code that benefits from UPC intranode shared memory communication, which is maximized on 64 and 128 cores.

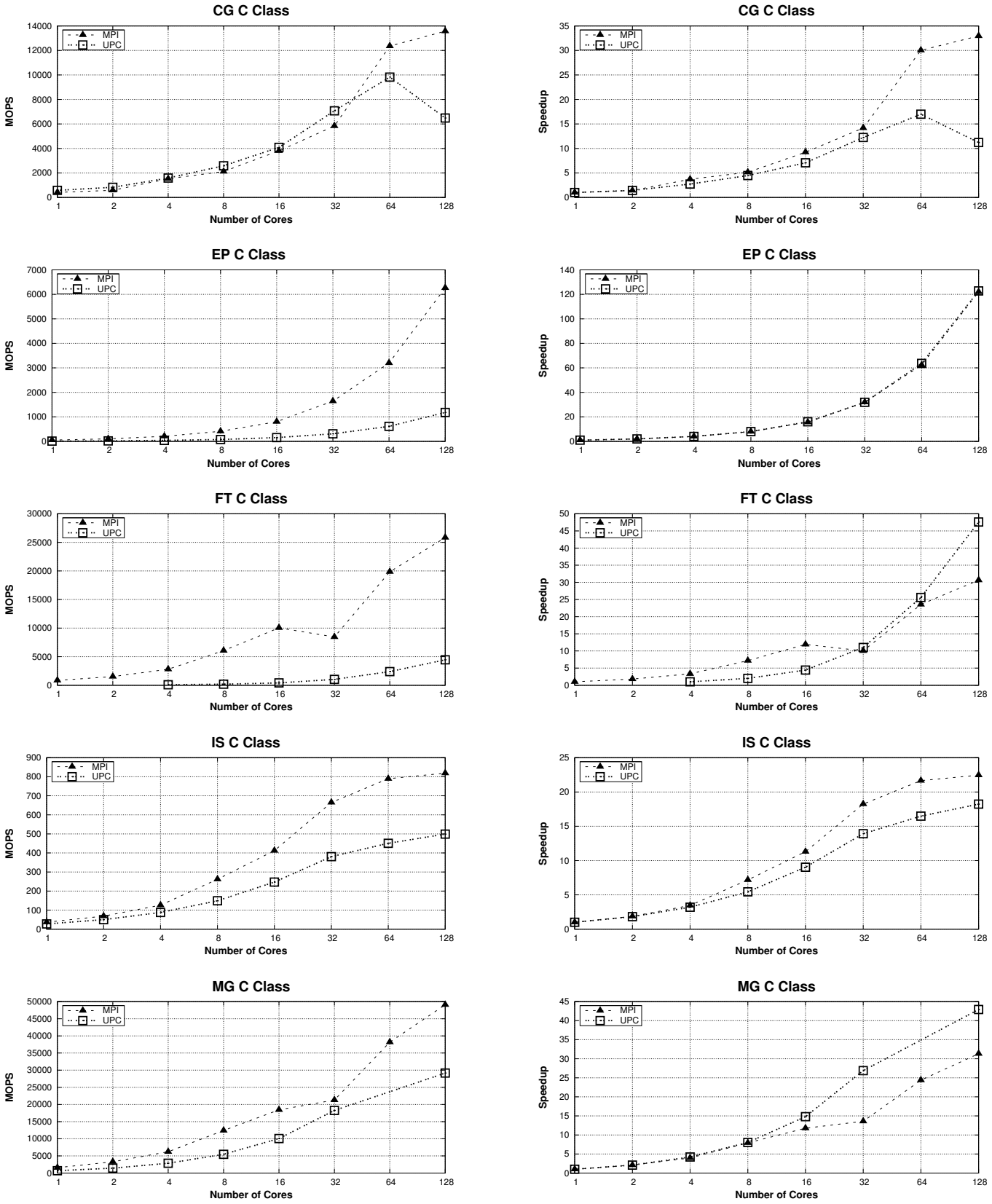

Figure 2. Performance of NPB kernels

IS is a quite communication-intensive code. Thus, both MPI and UPC obtain low speedups for this kernel (less than 25 on 128 cores). Although UPC IS has been optimized using privatization, the lower performance of its communications limits its scalability, which is slightly lower than MPI speedups.

Regarding MG, MPI outperforms UPC in terms of MOPS, whereas UPC shows higher speedup. The reason is, like in FT, the poor performance of UPC MG on 1 core, which allows it to obtain almost linear speedups on up to 16 cores.

V. Conclusions

This paper has presented two up-to-date performance evaluations: (1) two UPC collective libraries, being both compared with MPI; (2) UPC vs MPI in computational kernels.

The main conclusions of this work are: (1) there is a lack of collectives primitives benchmarks, so we have implemented our own UPC collectives microbenchmark suite, which is similar to a widely spread suite for MPI collectives (Intel MPI microbenchmarks); (2) the two collective implementations evaluated present significant differences in performance, which depends on the memory architecture, the message size and the communication pattern of the primitive; (3) it is possible to achieve important performance increases by automatically selecting the best collective primitive implementation at runtime; (4) UPC primitives take advantage of the use of hybrid shared/distributed memory configurations, currently the most commonly deployed ones (e.g., multicore clusters); (5) inefficient implementations of communications primitives have been detected, such as REF reduce; (6) UPC obtains quite poor collective performance compared to MPI; moreover, the best comparative results are obtained for long messages, as UPC suffers from high start-up communication latencies. This comparative evaluation shows that there is room for performance improvement in UPC collectives libraries.

Finally, it can be concluded that UPC codes can take full advantage of the use of efficient and scalable collectives primitives. Thus, the characterization of their performance is highly important. Furthermore, the higher programmability provided by collectives primitives, together with their locality exploitation, improves the productive development of efficient parallel applications in UPC.

As future work we intend to develop a more efficient UPC collective library that would take into account the locality and the communication overhead among all threads. Thus, in a hybrid shared/distributed memory architecture this library would minimize the number of remote memory operations. Moreover, the shared-memory (local) accesses can be improved by taking advantage of the affinity of UPC threads in order to improve the memory throughput.

Acknowledgments

This work was funded by Hewlett-Packard and partially supported by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02 and by the Galician Government (Xunta de Galicia, Spain) under the Consolidation Program of Competitive Research Groups (Ref. 3/2006 DOGA 13/12/2006). We gratefully thank Jim Bovay at HP for their valuable support, and CESGA for providing access to the Finis Terrae supercomputer.

References

[1] George Washington University, "Unified Parallel C at GWU," http://upc.gwu.edu [Last visited: August 2009].

[2] E. Wiebel, D. Greenberg and S. R. Seidel, "UPC Collective Operations Specifications v1.0," http://www.gwu.edu/upc/docs/UPC_Coll_Spec_V1.0.pdf [Last visited: August 2009].

[3] UPC Consortium, "UPC Language Specifications v1.2. (May 31, 2005)." http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf [Last visited: August 2009].

[4] "Berkeley UPC," http://upc.lbl.gov/ [Last visited: August 2009].

[5] Michigan Tech, "Collectives Reference Implementation," http://www.upc.mtu.edu/collectives/col1.html [Last visited: August 2009].

[6] Hewlett-Packard Inc., "HP Unified Parallel C (HP UPC)," http://hp.com/go/upc/ [Last visited: August 2009].

[7] IBM, "IBM XL Unified Parallel C (IBM XL UPC)," http://www.alphaworks.ibm.com/tech/upccompiler [Last visited: August 2009].

[8] NASA Advanced Computing Division, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Software/NPB/ [Last visited: August 2009].

[9] Z. Zhang and S. Seidel, "Benchmark Measurements of Current UPC Platforms," in *Proc. 4th Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'05)*, Denver (CO), 2005, p. 276b (8 pages).

[10] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: a NPB Experimental Study," in *Proc. 15th ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimore (MD), 2002, p. 1–26.

[11] T. El-Ghazawi and F. Cantonnet and Y. Yao and S. Annareddy and A. S. Mohamed, "Benchmarking Parallel Compilers: A UPC Case Study," *Future Generation Computer Systems*, vol. 22, no. 7, pp. 764–775, 2006.

[12] T. El-Ghazawi and S. Chauvin, "UPC Benchmarking Issues," in *Proc. 30th Intl. Conference on Parallel Processing (ICPP'01)*, Valencia (Spain), 2001, pp. 365–372.

[13] R. A. Salama and A. Sameh, "Potential Performance Improvement of Collective Operations in UPC," *Advances in Parallel Computing*, vol. 15, pp. 413–422, 2008.

[14] D. Bonachea, "UPC Collectives Value Interface, v1.2," http://upc.lbl.gov/docs/user/README-collectivev.txt [Last visited: August 2009].

[15] Z. Ryne and S. Seidel, "Ideas and Specifications for the new One-sided Collective Operations in UPC," http://www.upc.mtu.edu/papers/OnesidedColl.pdf [Last visited: August 2009].

[16] Z. Ryne and S. Seidel, "UPC Extended Collective Operations Specification," http://www.upc.mtu.edu/papers/UPC_CollExt.pdf [Last visited: August 2009].

[17] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without Pain = Productivity: Data Layout and Collective Communication in UPC," in *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City (UT), 2008, pp. 99–110.

[18] D. Bonachea, "Proposal for Extending the UPC Memory Copy Library Functions, v2.0," http://upc.lbl.gov/publications/upc_memcpy.pdf [Last visited: August 2009].

[19] Intel Corporation, "Intel MPI Benchmarks," http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm [Last visited: August 2009].

[20] W. Gropp and E. Lusk, "Reproducible Measurements of MPI Performance Characteristics," in *Proc. 6th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'99). Lecture Notes in Computer Science vol. 1697*, Barcelona (Spain), 1999, pp. 11–18.

[21] T. A. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: a NPB Experimental Study," in *Proc. of the 15th ACM/IEEE Conf. on Supercomputing (SC'02)*, Baltimore (MD), 2002, pp. 1–26.

[22] "Finis Terrae Supercomputer," http://www.top500.org/system/details/9500 [Last visited: August 2009].